DISTRIBUTED SYSTEMS

CONCEPTS AND DESIGN

second edition

GEORGE COULOURIS
JEAN DOLLIMORE
TIM KINDBERG

Queen Mary and Westfield College
University of London

ADDISON-WESLEY
Harlow, England • Reading, Massachusetts • Menlo Park, California
New York • Don Mills, Ontario • Amsterdam • Bonn • Sydney • Singapore
Tokyo • Madrid • San Juan • Milan • Mexico City • Seoul • Taipei

Addison Wesley Longman Limited
Edinburgh Gate
Harlow
Essex, CM20 2JE
England

Cover designed by Hybert Design and Type, Maidenhead
Printed in the United States of America

1st and 2nd impressions 1994.
3rd and 4th impressions 1995.
Reprinted 1996.
6th and 7th impressions 1998.

ISBN 0–201–62433–8

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

Library of Congress Cataloging-in-Publication Data is available

FORE

DISTRII
THE CH

by Kennet
Associate

I find it cur
years now,
it did when
approach di
sorts of app
important e:
approaches
areas, stan
confirming
bringing th
floor, the c
Yet, i
understood
computing
systems, th
The US go
computing
pervasive t
This
and Design
state-of-the
computing
general-pur
time, the b

14.4 Concurrency control in distributed transactions

Each server manages a set of data items and is responsible for ensuring that they remain consistent when accessed by concurrent transactions. Therefore each server is responsible for applying concurrency control to its own data items. The members of a collection of servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner. This implies that if transaction T is before transaction U in the serially equivalent ordering of transactions at one of the servers then they must be in that order at all of the servers whose data items are accessed by both T and U.

Locking in distributed transactions

In a distributed transaction, each server maintains locks for its own data items. The local lock manager can decide whether to grant a lock or make the requesting transaction wait. However it cannot release any locks until it knows that the transaction has been committed or aborted at all the severs involved in the transaction. When locking is used for concurrency control, the data items remain locked and are unavailable for other transactions during the atomic commit protocol, although an aborted transaction releases its locks after phase 1 of the protocol.

As servers set their locks independently of one another, it is possible that different servers may impose different orderings on transactions. In some cases, these different orderings can lead to cyclic dependencies between transactions and a distributed deadlock situation arises. The detection and resolution of distributed deadlocks is discussed in the next section of this chapter. When a deadlock is detected, a transaction is aborted to resolve the deadlock. In this case, the coordinator will be informed and will abort the transaction at the workers involved in the transaction.

In nested transactions, parent transactions are not allowed to run concurrently with their child transactions, to prevent potential conflict between levels. Nested transactions inherit locks from their ancestors. For a nested transaction to acquire a read lock on a data item, all the holders of write locks on that data item must be its ancestors. Similarly, for a nested transaction to acquire a write lock on a data item, all the holders of read and write locks on that data item must be its ancestors. When a nested transaction commits its locks are inherited by its parent. When a nested transaction aborts, its locks are removed.

Timestamp ordering concurrency control in distributed transactions

In a single server transaction, the server issues a unique timestamp to each transaction when it starts. Serial equivalence is enforced by committing the versions of data items in the order of the timestamps of transactions that accessed them. In distributed transactions, we require that each server is able to issue globally unique timestamps. A globally unique transaction timestamp is issued to the client by the first server accessed by a transaction. The transaction timestamp is passed to each server that performs an operation in the transaction.

The servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner. For example, if the version of a data

C

item accessed by transactic
then if T and U access the
commit them in the same
servers must agree as to the
<local timestamp, server-i
comparison in which the se

The same ordering o
local clocks are not synchr
the timestamps issued by c
by the other servers. Wh
corresponds to the order in
roughly synchronized by th

When timestamp orc
as each operation is perfor
aborted, the coordinator v
workers. Therefore any tr
always be able to commit
commit protocol will norm
will not agree to commit it

Optimistic concurrenc

Recall that with optimistic
allowed to commit. Serv
transactions are serialize
distributed transaction is
validates transactions that
takes place during the firs

Consider the follo
items A and B at servers

T
Re
Wr
Re
Wr

The transactions access t
U before T at server Y.
time, but server X valida
recommends a simplific
transaction may perform
server will be unable to
This is an example of cc

item accessed by transaction U commits after the version accessed by T at one server, then if T and U access the same data item as one another at other servers, they must commit them in the same order. To achieve the same ordering at all the servers, the servers must agree as to the ordering of their timestamps. A timestamp consists of a pair *<local timestamp, server-id>*. The agreed ordering of pairs of timestamps is based on a comparison in which the server-id part is less significant.

The same ordering of transactions can be achieved at all the servers even if their local clocks are not synchronized. However for reasons of efficiency it is required that the timestamps issued by one server should be roughly synchronized with those issued by the other servers. When this is the case, the ordering of transactions generally corresponds to the order in which they are started in real time. Timestamps can be kept roughly synchronized by the use of synchronized local physical clocks (see Chapter 10).

When timestamp ordering is used for concurrency control, conflicts are resolved as each operation is performed. If the resolution of a conflict requires a transaction to be aborted, the coordinator will be informed and it will abort the transaction at all the workers. Therefore any transaction that reaches the client request to commit should always be able to commit. Therefore a server involved as a worker in the two-phase commit protocol will normally agree to commit. The only situation in which a worker will not agree to commit is if it had crashed during the transaction.

Optimistic concurrency control in distributed transactions

Recall that with optimistic concurrency control, each transaction is validated before it is allowed to commit. Servers assign transaction numbers at the start of validation and transactions are serialized according to the order of the transaction numbers. A distributed transaction is validated by a collection of independent servers each of which validates transactions that access its own data items. The validation at all of the servers takes place during the first phase of the two-phase commit protocol.

Consider the following interleavings of transactions T and U that access data items A and B at servers X and Y respectively.

T		*U*	
Read(A)	at X	*Read(B)*	at Y
Write(A)		*Write(B)*	
Read(B)	at Y	*Read(A)*	at X
Write(B)		*Write(A)*	

The transactions access the data items in the order T before U at server X and in the order U before T at server Y. Now suppose that T and U start validation at about the same time, but server X validates T first and server Y validates U first. Recall that Chapter 13 recommends a simplification of the validation protocol that makes a rule that only one transaction may perform validation and write phases at the same time. Therefore each server will be unable to validate the other transaction until the first one has completed. This is an example of commitment deadlock.

The validation rules in Chapter 13 assume that validation is fast which is true for single server transactions. However in a distributed transaction, the two-phase commit protocol may take some time and will delay other transactions from entering validation until a decision on the current transaction has been obtained. A parallel validation protocol is generally used to increase the concurrency at each individual server. Kung and Robinson [1981] describe parallel validation in their original paper. It must check conflicts between write operations of the transaction being validated against the read operations of other concurrent transactions.

If parallel validation is used, transactions will not suffer from commitment deadlock. However if servers simply perform independent validations, it is possible that different servers of a distributed transaction may serialize the same set of transactions in different orders. For example with T before U at server X and U before T at server Y in our example.

The servers of distributed transactions must prevent this from happening. One approach is that after a local validation by each server, a global validation is carried out [Ceri and Owicki 1982]. The global validation checks that the combination of local orderings at the individual servers is serializable. That is, that the transaction being validated is not involved in a cycle.

Another approach is that all of the servers of a particular transaction use the same globally unique transaction number at the start of the validation [Schlageter 1982]. The coordinator of the two-phase commit protocol is responsible for generating the global unique transaction number and passes it to the workers in the *CanCommit?* message. As different servers may coordinate different transactions, the servers must (as in the distributed timestamp ordering protocol) have an agreed order for the transaction numbers they generate.

14.5 Distributed deadlocks

The subsection on deadlocks in Section 13.2 shows that deadlocks can arise within a single server when locking is used for concurrency control. Servers must either prevent or detect and resolve deadlocks. Using timeouts to resolve possible deadlocks is a clumsy approach – it is difficult to choose an appropriate timeout interval so that transactions are aborted unnecessarily. With deadlock detection schemes, a transaction is aborted only when it is involved in a deadlock. Most deadlock detection schemes operate by finding cycles in the transaction wait-for graph. In a distributed system involving multiple servers being accessed by multiple transactions, a global wait-for graph can in theory be constructed from the local ones. There can be a cycle in the global wait-for graph that is not in any single local one – that is, there can be a **distributed deadlock**. Recall that the wait-for graph is a directed graph in which nodes represent transactions and data items; and edges represent either a data item held by a transaction or a transaction waiting for a data item. There is a deadlock if and only if there is a cycle in the wait-for graph.

Figure 14.11 shows the interleavings of the transactions U, V and W involving data items A and B managed by servers X and Y and data items C and D managed by server Z.

Fig 14.11 Interleavi

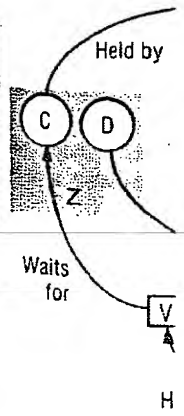
Deposit(

Deposit(

Withdra

The
consists o
data item
at a time,
Det
transactio
the transa

Fig 14.12 Distribute



H

(a)